Software Architectures

Distributed Systems

Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2012/2013

Outline

These Slides Contain Material from [Tanenbaum and van Steen, 2007]

Slides were made kindly available by the authors of the book

- Such slides shortly introduced the topics developed in the book [Tanenbaum and van Steen, 2007] adopted here as the main book of the course
- Most of the material from those slides has been re-used in the following, and integrated with new material according to the personal view of the teacher of this course
- Every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of the teacher of this course

Outline

Software Architectures to Handle Complexity

Distributed systems are complex

- In order to manage their intrinsic complexity, distributed systems should be properly organised
- Organisation of a distributed system is mostly expressed in terms of its software components

Software architectures expresses component organisation

- Many ways to organise components of a distributed system, classified as *software architectures*
- Many instantiations where components have their actual placed in a distributed system—often called *system architectures*

Architectural Style

An architectural style is formulated in terms of...

- components
- the way in which components are connected to each other
- the data flowing through the components
- the way in which all the above things are configured altogether to build the system

The notion of architectural style...

- encompasses a way to cluster and classify groups of similar systems, that is, having the same sort of organisation
- allow distributed systems to be compared
- but also provide general patterns for their overall design

Components & Connectors I

Components

- A component is a modular unit with well-defined *interfaces*
- which is *replaceable* within its environment
- interfaces are both *required* and *provided*—both ways, then

Connectors

- A connector is an abstraction *mediating* communication, coordination, cooperation among components
- that is, anything providing a *mechanism for interaction* among components

Components & Connectors II

Putting together components and connectors

- . . . produces a huge range of possible organisations and configurations
- that are then classified in terms of architectural styles

Architectural Styles for Distributed Systems

Identification of architectural styles

- Architectural styles – like patterns in software engineering – are to be devised out rather than invented
- Today, four different architectural styles have been identified as the main ones for distributed systems

Important styles of architecture for distributed systems

- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

Layered Architectures

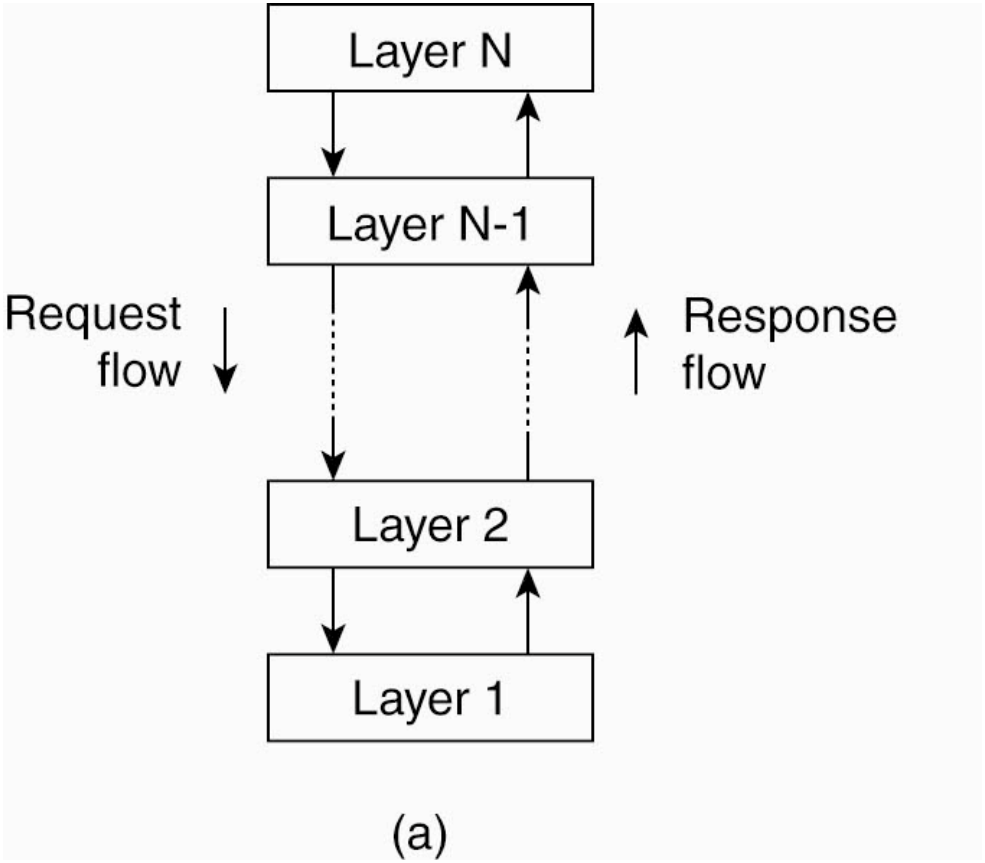
Basic idea

- Components are organised in a *layered fashion*
- where components of a layer *only* call components of the layer below, and are *only* called by the components of the layer above

Data flow

- The request-response flow is always top-down / bottom-up
- Control flow follow the same pattern along with data

Layered Architecture Style



[Tanenbaum and van Steen, 2007]

Object-based Architectures

Basic idea

- Components are objects
- Components are connected through a RPC mechanism

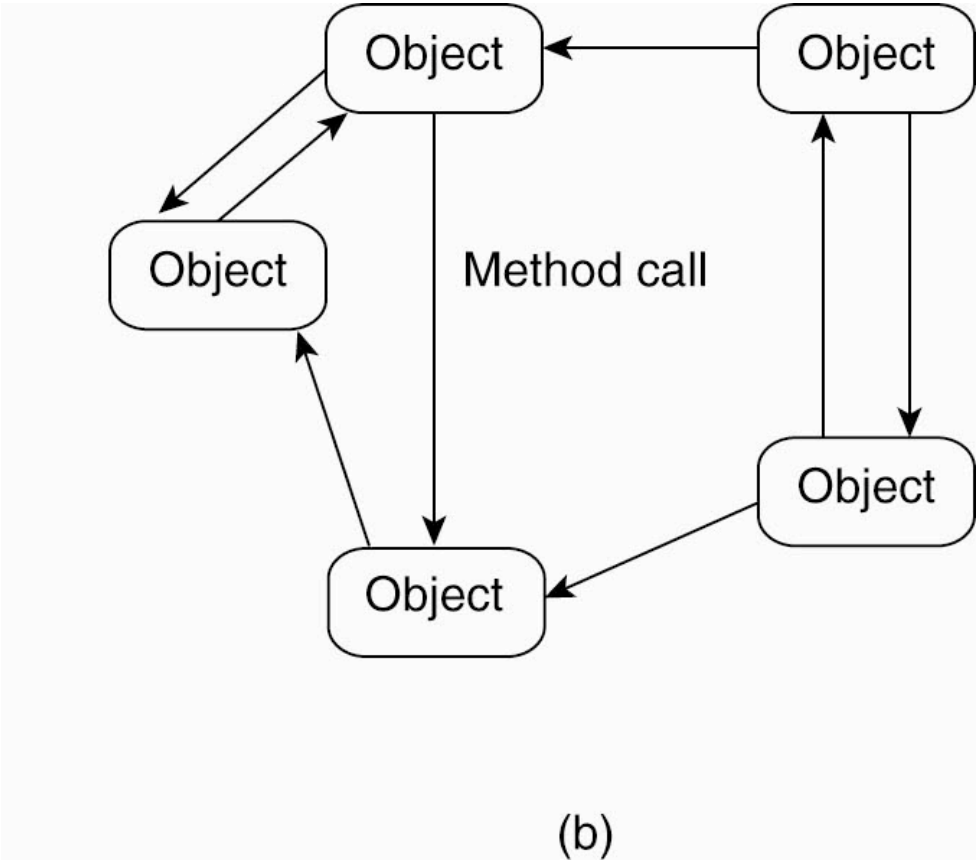
Client-server architectures

- . . . are built out of this style

Layered and object-based architectures

- are the most important styles for distributed systems today
- However, a lot of things are going to happen in the future, which may change such an overall picture

Object-based Architecture Style



[Tanenbaum and van Steen, 2007]

Data-centred Architectures I

Basic idea

- Communication among processes occurs through a shared repository
- The repository might be either passive (reactive) or (pro)active

Main features

- ... depends on the choice made for the shared repository
- how information is represented
- how events are handled
- how the shared repository behave in response to interaction
- how processes interact with / through the shared repository

Data-centred Architectures II

Examples are everywhere

- Web-based systems, for instance, are largely data-centric
- Also, many distributed applications still work by sharing files around the network

Event-based Architectures

Basic idea

- Processes communicate through an event bus
- through which events are propagated
- possibly carrying data along

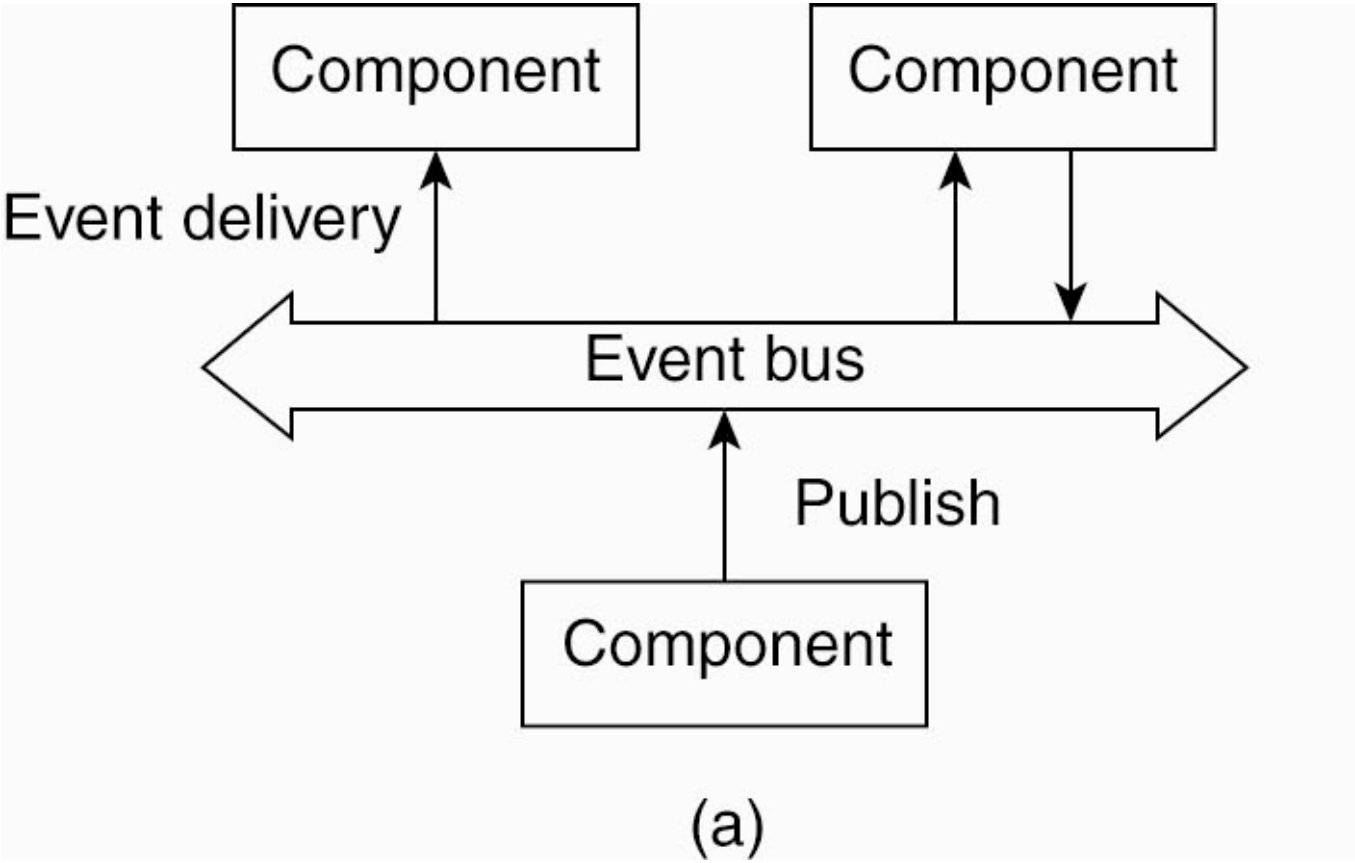
Main example: Publish / subscribe systems

- Publishers publish events through the middleware
- Subscribers receive events to which they have subscribed

Main feature

- Processes can communicate with no need of reference each other / to know each other, they are *referentially decoupled*
- Processes can communicate with no need to share the same space, they are *decoupled in space*

Event-based Architecture Style



[Tanenbaum and van Steen, 2007]

Shared Data-space Architectures I

Basic idea

- Putting together Data-centric and Event-based architectures
- The shared repository is a shared persistent data-space, and also an event bus
- where data is stored and accessed
- along with related events

Main example: Blackboard systems

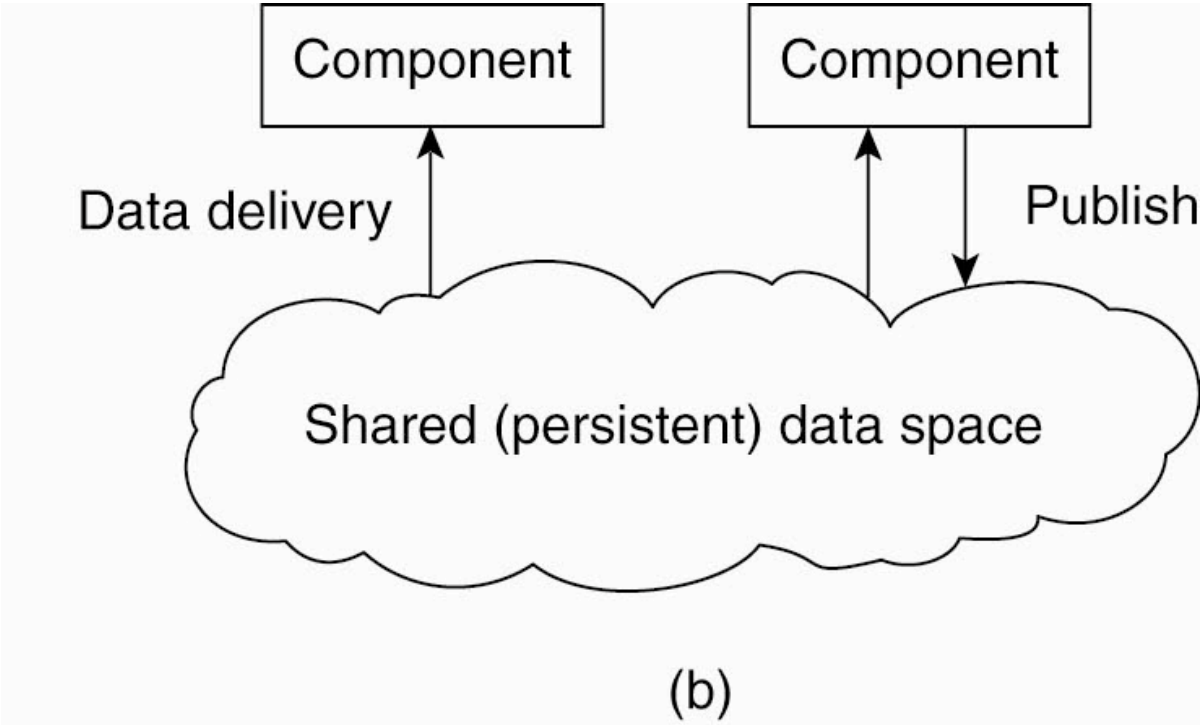
- Processes put data in the blackboard
- The blackboard aggregates knowledge, implements policies and drive the coordination of processes

Shared Data-space Architectures II

Main feature

- Processes can communicate with no need of compresence
- Processes are also *decoupled in time*

Shared Data-space Architecture Style



[Tanenbaum and van Steen, 2007]

Outline

Where are Software Components?

Component Topology

- When a software architecture is actually instantiated, components are placed somewhere in a distributed system
- This is typically taken as an instantiation of a software architecture in a *system architecure*

Sorts of System Architectures

- Centralised architectures
- Decentralised architectures
- Hybrid architectures

Outline

Clients & Servers

Main feature

- In a centralised architecture, *clients* request *services* from *servers*—and that is all, more or less
- In the basic client-server model, processes are classified in two groups—obviously, clients and servers
- Possibly, the two groups may overlap

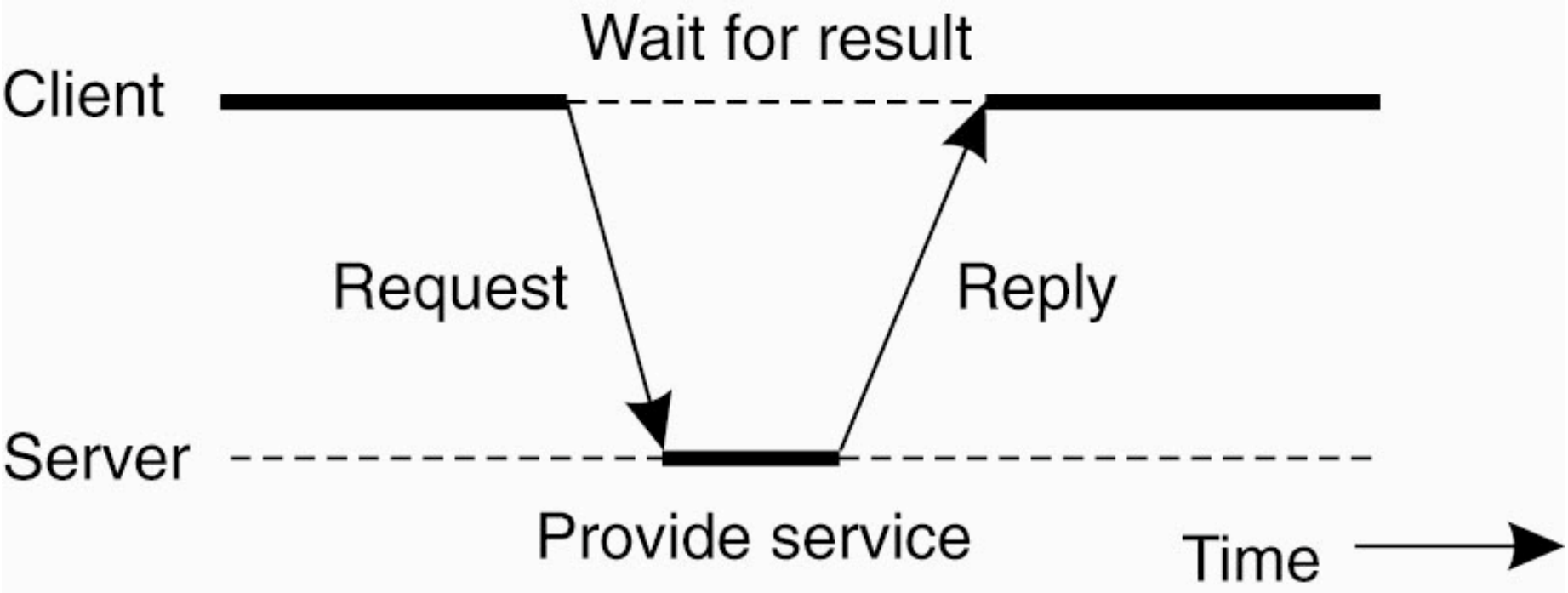
Servers

A server is a process implementing a specific service—like, say, a database service

Clients

A client is a process requiring a specific service from a server

Client-server Interaction



Scheme of client-server interaction: *request-reply behaviour*
[Tanenbaum and van Steen, 2007]

Client-Server Communication

Efficiency vs. reliability

- Connectionless protocols is ok for *idempotent* operations
  - that is, operations that could be repeated more than once without harm
- Connection-oriented protocols are less efficient, but ensure reliability
- For instance, Internet protocols are typically based on TCP/IP connections—reliable but relatively costly for small-grain communication

Application Layering

Logical layering in client-server architectures

User-interface level contains the interface with the user

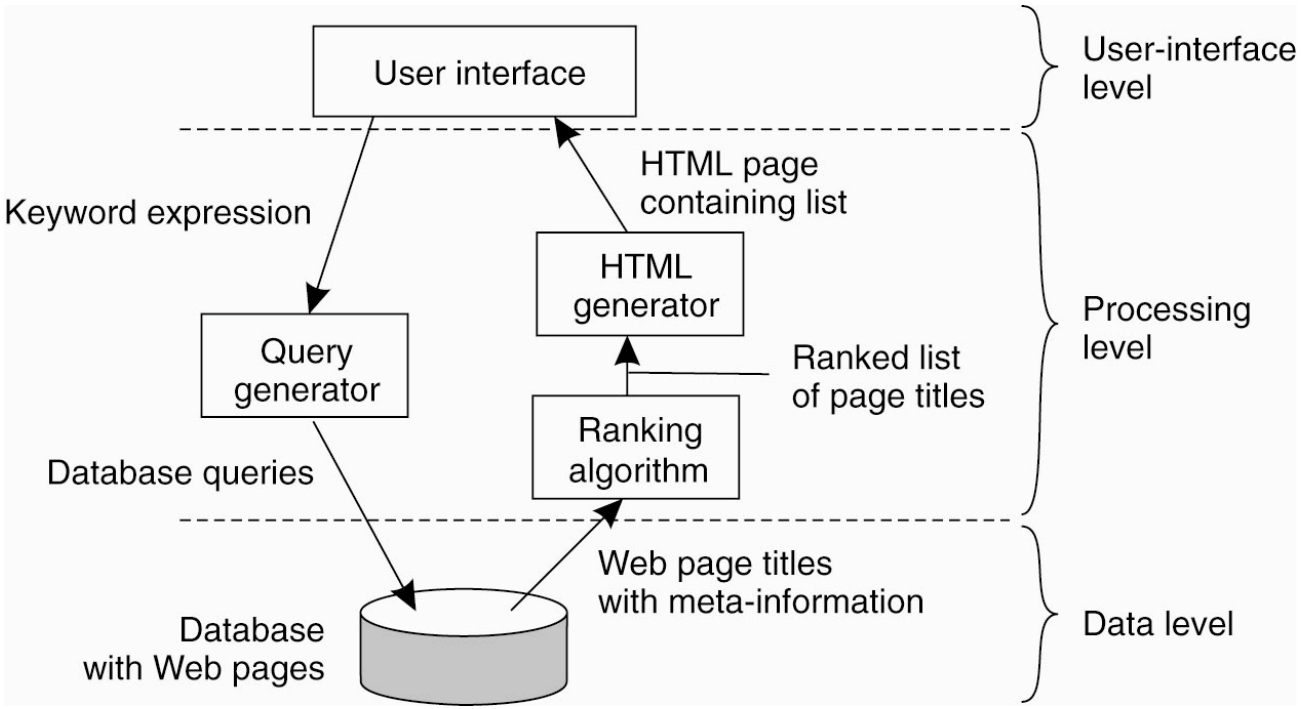
Processing level contains the logic of the control, in short, the core of the applications

Data level manages the actual data that are relevant to the applications

Typical organisation for client-server applications

- with a part handling user interaction,
- a part dealing with data and files,
- and a part containing the core functionality of an application

Example: Internet Search Engine



The simplified organisation of an Internet search engine into three different layers

[Tanenbaum and van Steen, 2007]

Multi-tiered Architectures

How to physically distribute logical layers?

- Logical organisation is not physical organisation
- Clients and servers could be placed on the same node, or be distributed according to several different topologies

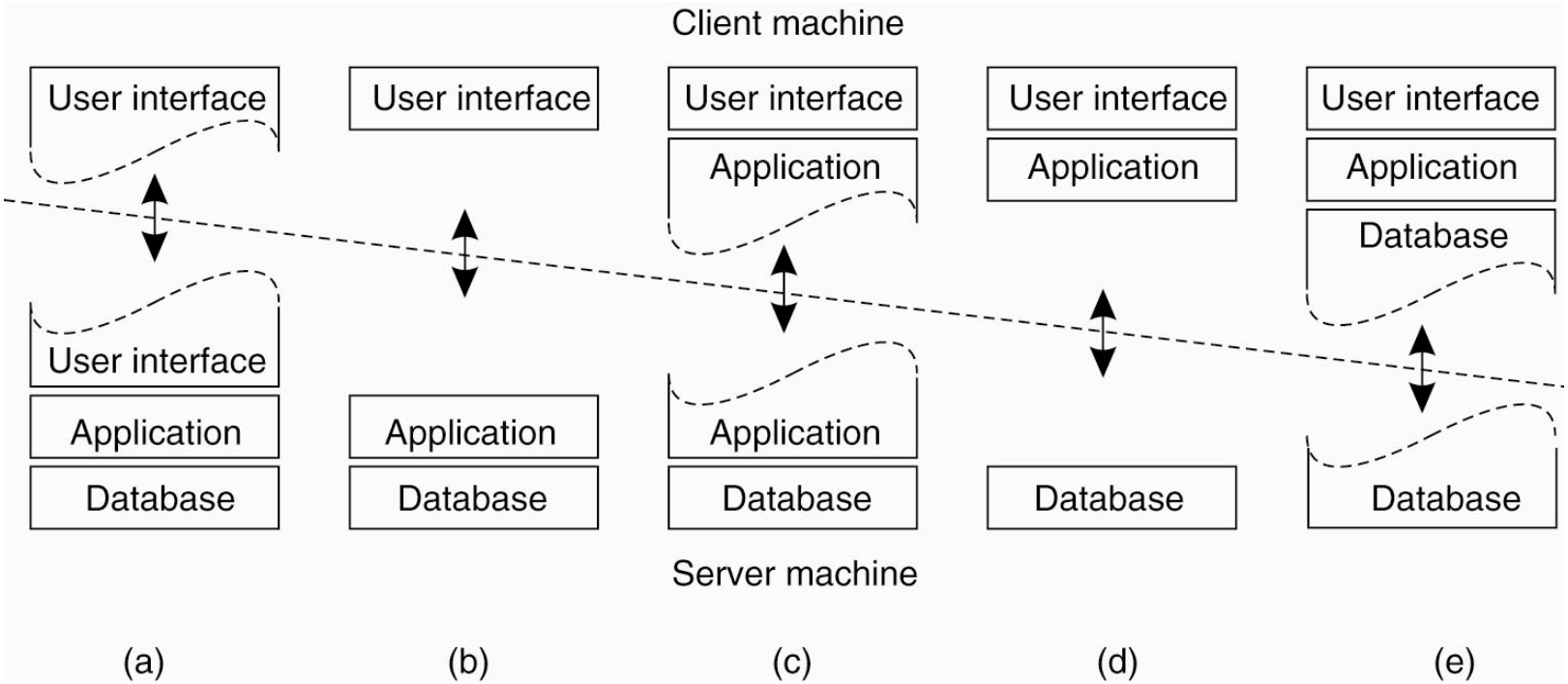
Two-tiered architecture

- The simplest choice is to have just two sort of machines
- hosting either servers or clients
- resulting in the (physically) two-tiered architecture

Choices for two-tiered architecture

- Where are the three application-layers placed?
- On the client machines, or on the server machines?
- a range of possible solutions, accordingly

Possible Two-tiered Organisations



Alternative client-server organisations
[Tanenbaum and van Steen, 2007]

Current Trends in Two-tiered Architectures

Moving toward the clients

- Scalability pushes charge far from servers
- Along with more efficient network connections, more powerful client machines, and above all more expressive technologies for distributing applications

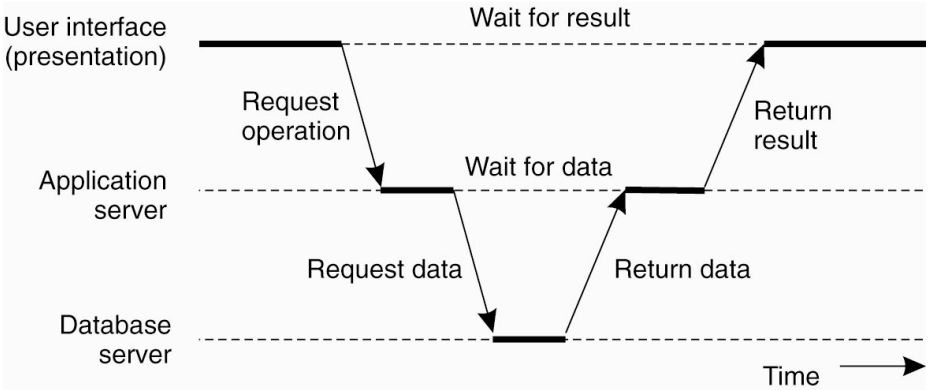
Thin vs. fat clients

- Thin clients are simpler
- Fat clients are more complex, but are typically more efficient from the user's viewpoint, and more scalable from the engineer's viewpoint

Three-tiered Architectures

Servers may sometimes act as clients

- Servers might be layered, in turn
- We may (physically) distinguish between application servers and database servers
- Example: the Transaction Processing Monitor discussed in the previous lessons



An example of a server acting as client [Tanenbaum and van Steen, 2007]

Outline

Vertical vs. Horizontal Distribution I

Vertical distribution

- Multi-tiered client-server architectures directly derive from the three levels of applications
- Logical organisation is mapped onto the tiers
- Often, distributed processing amounts at building a client-server application according to a multi-tiered architecture
- This is typically called *vertical distribution*

Vertical vs. Horizontal Distribution II

Horizontal distribution

- Sometimes, the physical distribution of the clients and the servers is what actually counts
- Clients and servers may be physically split into logically-equivalent parts, each one working on its own portion of the whole data set
- This is typically called *horizontal distribution*
- This is an obviously decentralised class of systems

Horizontal Distribution: Main Example I

Peer-to-peer systems

- All the processes in a peer-to-peer system are equal
- So, every process works to the system main function, whatever it is
- Each process works then at the same time as a client and as a server
- So, it is typically called *servent*

Overlay network

- Peer-to-peer architectures are symmetric
- So, the main problem of peer-to-peer architectures is how to organise the network whose nodes are the servents and the links are the communications among them
- Such a network organisation is typically called an *overlay network*

Horizontal Distribution: Main Example II

Types of overlay networks

- Processes communicate through available communication channels
- Overlay networks may be either structured or unstructured
- Accordingly, the two main sorts of peer-to-peer architectures are
  - Structured peer-to-peer architectures
  - Unstructured peer-to-peer architectures

Outline

Combining the Benefits

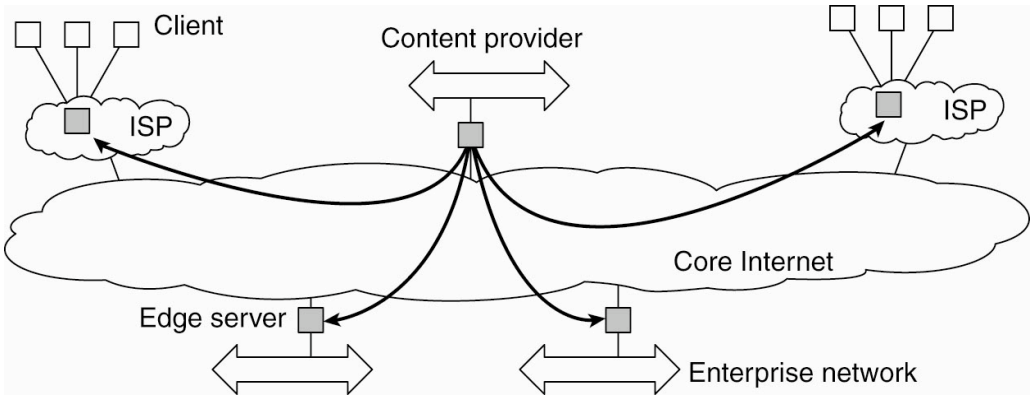
Hybrid architectures

- Many distributed systems require properties from both client-server and peer-to-peer architectures
- So, they put together features from both centralised and decentralised architectures
- These are typically called *hybrid architectures*

Edge-Server Systems

Servers are "on the edge" of the network

- The "edge" is formed by the boundary between the enterprise network and the actual Internet
- For instance, home clients connecting through an ISP (Internet Service Provider)



Viewing the Internet as consisting of a collection of edge servers [Tanenbaum and van Steen, 2007]

Collaborative Distributed Systems I

Main idea

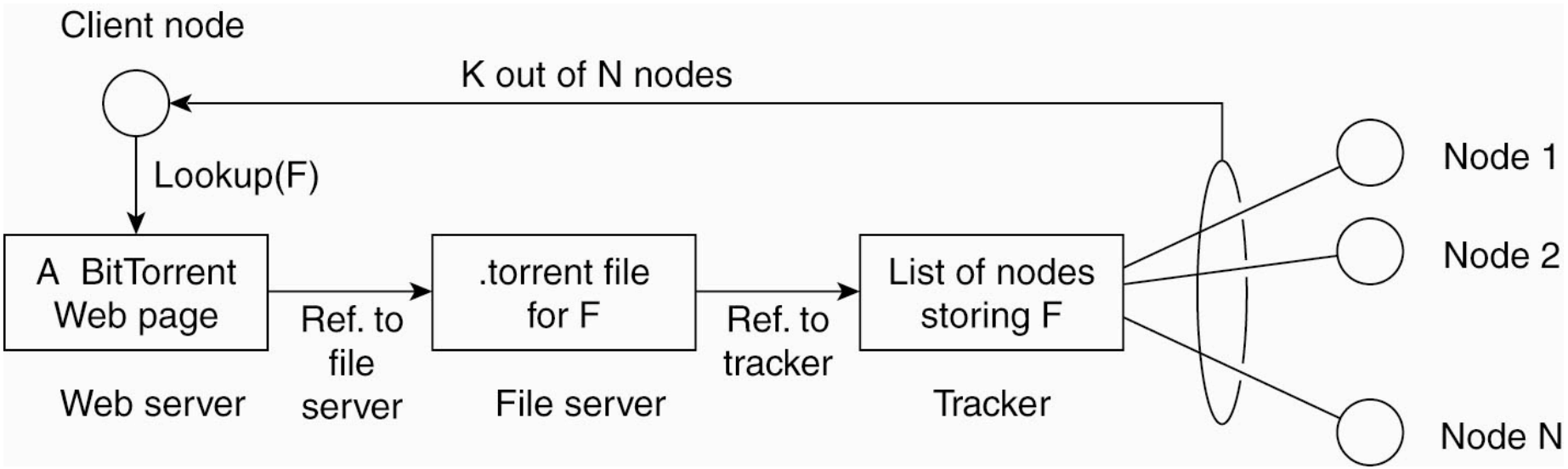
- The main problems of these systems is to get started: a traditional client-server scheme is then used here
- Once a node has joined the system, collaboration proceeds using a fully decentralised scheme

Collaborative Distributed Systems II

Main example: BitTorrent

- BitTorrent is a peer-to-peer file downloading system
- When a user needs a file in BitTorrent, he/she gets chunks of the file from other users around until he/she gets it all
- A file can be downloaded by a client only when the client is providing files to other clients
- A global directory provides *.torrent* files that points to the *trackers*
- Trackers are servers knowing *active*, collaborating nodes that can provide the requested chunks
- Collaboration of nodes is promoted by suitable reward / punishment policies

BitTorrent as a Collaborative Distributed System



The principal working of BitTorrent
[Tanenbaum and van Steen, 2007]

Outline

Which Middleware for Which Architecture? I

Main problem

- In practice, middleware commonly incorporates some architectural element / abstraction / component / style
- For instance, CORBA is designed around the object-oriented architectural style
- This means that middleware tends to be not adaptable to every application scenario
- The solution of adding different abstractions and elements affects conceptual integrity of middleware and of the resulting applications

Which Middleware for Which Architecture? II

The typical solution

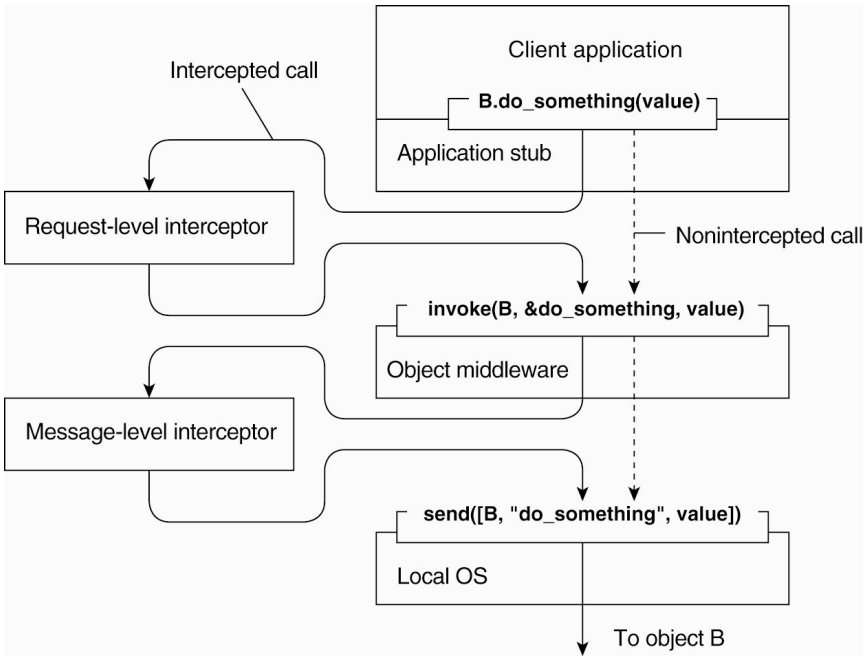
- As usual and as generic as it may seem, it is again separating mechanisms from policies
- This allow the behaviour of the middleware to be modified according to the application needs

Interceptors

Main idea

- A software construct
- Intercepting the normal flow of control
- Allowing policies to be added that are application-specific

Using interceptors to handle remote-object invocations [Tanenbaum and van Steen, 2007]



Outline

Adapting Middleware

Main idea

- The problem of (unpredictable) change
- Any fixed solution / response may fail when facing an unpredictable modification
- E.g., interceptors represent a generic solution to adaptation in terms of a naive mechanism

Adaptive software?

- Easier said than done
- Preparing for the unpredictable might result quite an issue, indeed
- Said that, this is one of the hottest fields of research in computer science

Toward Adaptive Software

Three basic techniques [McKinley et al., 2004]

- Separation of concerns
- Computational reflection
- Component-based design

Toward Adaptive Software

Three basic techniques [McKinley et al., 2004]

- Separation of concerns
- Computational reflection
- Component-based design

Separation of concerns

Separating functional and non-functional

- Non-fuctional properties like reliability, performance, security, . . . , should be faced separatedly
- ????
- OK, forget about this, this does not work really
- *Aspect-oriented programming* and *aspect-oriented software development* deals with cross-cutting concerns

Toward Adaptive Software

Three basic techniques [McKinley et al., 2004]

- Separation of concerns
- Computational reflection
- Component-based design

Computational reflection

The ability to inspect oneself and possibly self-adapt behaviour

- Reflection if at the core of modern programming language like Java
- Observing the state of a program by the program itself
- Reification is changing the state of the program after reflection
- Observing oneself state related with the environment makes it possible to change behaviour adaptively

Toward Adaptive Software

Three basic techniques [McKinley et al., 2004]

- Separation of concerns
- Computational reflection
- Component-based design

Component-based design

Adaptation through composition

- Once an architecture is open—e.g., hot-pluggable
- A new behaviour may be added by adding a component on the fly
- Once an architecture for open systems is available, the point is how to select a component that may add the required behaviour to the system

Outline

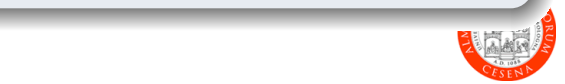
Automatic Adaptation

Main idea

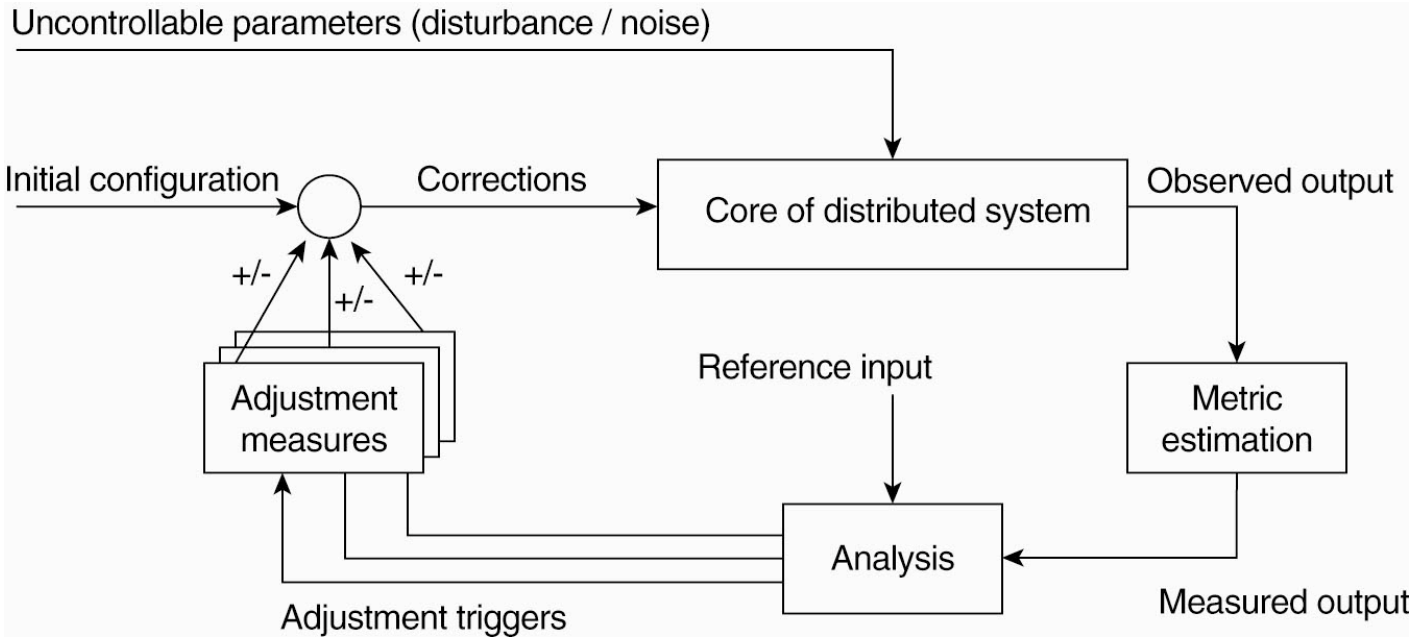
- Unpredictability of change makes guided adaptation essentially faulty
- Systems should be able to detect (relevant) change in the environment and consequently change / adapt
- This is the field of *autonomic computing* [Kephart and Chess, 2003] and of *self-* systems* [Babaoglu et al., 2005]

Many views on self-* systems

- What all of them have in common is that adaptations come from some *feedback loop* of some sort
- Including some perception of the environment and of its change in the loop



The Feedback Control Model



Feedback control model: Logical organisation
[Tanenbaum and van Steen, 2007]

Summing Up I

Organisation of distributed systems

- Software architectures and system architectures deal with software organisation
- They are approximative and maybe non-scientific ways to model systems
- However they are expressive and abstract enough to help distributed system engineering

Summing Up II

Main issues

- Software architectures are concerned with logical organisation
- System architectures are concerned with component placement in a distributed setting
- Adaptation is a must in modern and forthcoming systems
- Autonomic computing and self-* systems are at the edge of research in distributed systems nowadays

References I

Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., and van Steen, M., editors (2005).
Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations, volume 3460 of *Lecture Notes in Computer Science*.
Springer.

Kephart, J. O. and Chess, D. M. (2003).
The vision of autonomic computing.
IEEE Computer, 36(1):41–50.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. (2004).
Composing adaptive software.
IEEE Computer, 37(7):56–64.

References II

Tanenbaum, A. S. and van Steen, M. (2007).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.

Software Architectures

Distributed Systems

Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2012/2013